Assignment 2 Report

Siyamthanda Phakathi

PHKSIY006

Submitted to the University of Cape Town

In accordance to the institutions Academic

Dishonesty Policy

Faculty of Science

CSC 2001F

March 2025

Object Oriented Design

For this assignment I created 4 classes and 1 python script. These in conjunction with the the classes provided by Professor Hussein Suleman allowed me to fulfil the needs of this project. Below are the classes I created and their functionality.

Entry.java is an entry object where the term, statement and confidence score are all properties of the object. Its implementation is the same as that used in assignment 1. It consists of basic getters and setters and a compareTo method to compare entries by term. It also has a basic toString method.

The AVLTree has been constructed using the classes provided by professor Hussein Suleman.

KnowledgeBaseAVL.java is the backend for all tree manipulation and txt processing. Two AVL trees, namely entries and queries, are initialized.

- AddEntries(fileName) takes in the name of a txt file such as GenericsKB.txt and tokenizes the term, statement and confidence score. It then used those values to create a new Entry object and add it to the entries tree. This function is called assuming the tree is still empty. (For counting reasons, will be explained later)
- AddQueries also reads in a txt file full of terms to search for into the queries tree.
- appendEntries() works the same as AddEntries however our insertCounts for entries is reset to 0 as to count the time taken to add new entries to a tree. Variables insertCounts and findCounts are initialized upon creation in BinaryTree class and are set to 0. They are incremented everytime there is a comparison by term for the insert and find function in AVLTree.
- Search(query) simply searches for a term in the tree. If found it is printed out otherwise term not found is printed out.
- SearchAll() recursively searches for all terms in the queries tree one by one. Printing out if it is found or not. It starts searching from the root node
- getInsertOperationsCount() returns the number of insert operations done so far.
- getSearchOperationCount() returns the number of search operatins done so far.

AVLTreeTest.java is used for part one. It takes in a premade file of 10 entries (EntriesTest.txt) and 10 queries(QueriesTest.txt) and tests all the functions of the knowledge base and prints out the results to ResultsEntriesTest.txt. You can run it from the terminal using **make test**

GenericsKbAVLApp.java takes in a randomised data file of size n and loads it to our entries tree through the knowledge base. It then takes in a randomised query file and a randomised inserts file of size m. We will first search for the queries and note the search time and then append the tree with our inserts file. This will measure the time taken to insert m entries into a tree of size n. Output is printed to a Results text file. To run manually with your own files use the following command substituting the variables for the actual file names:

java -cp bin GenericsKbAVLApp dataFile queryFile insertsFile

Experiment

In this experiment we are testing the time complexities for insertions and searches in an AVL tree. We want to see how varying the size of the dataset n will effect our numbers of comparison. To do so we will use 10 values of n spaced equally apart logarithmically. One run of our testing process will go as follows:

- Create a subset of m entries that are not in GenericsKB.txt to be inserted called Entries{m}.
- Create a subset of m queries from GenericsKB-queries.txt to be searched called Queries{m}.
- These created files will be used for all values of n as to measure the effect of changing our dataset not our insertions and queries.
- Create a subset of n entries from GenericsKB.txt called Generics{n}.txt for all values of n.
- Run GenericsKbAVLApp for all our n entries files (Generics{n}.txt) and for each we will use the same Entries and Queries subset to measure our time.
- Determine the best case, worst case and average of all the values we receive from this.
- We will then repeat the process above r times to ensure our numbers are more accurate over time.
- Once all these values have been collected plot a graph of all these values and compare with our theoretical complexity for insert and search.

All of this can be done manually but I have used a python script to automate the entire process. The code for the automated process will be explained in the creativity section below are the results and conclusions from the experiment.

Testing Part 1

For part 1 I used AVLTreeTest.java and used the following 10 queries and 10 entries. To run the test of loading and finding queries run **make test** in the terminal to test the loading of files and our instrumentation.

The results of the test are as follows:

From these results we know that all of our functions to add entries, queries and search for them all work as we included 5 queries that could be found and 5 that can't. From this we know our Knowledge Base works. Our instrumentation also works as the values are below our upper bound of $\log_2(10) \times 10 \approx 34$. We use base 2 as it is a binary tree.

Testing experiments

We will now test our full program using a randomised set of n as outlined under the experiments heading. Note that for this program we have fixed the size of our Queries10 and Entries10 to 10 entries. Therefore the only variance will come from our sample generics. Our python script automatically runs our program and stores the results of each run for each n in **Results.txt**. The values below and those in our assignment are after our program has been ran 100 times and aggregated the results. Note that it is not recommended to run this many times as it will take long. For a relatively accurate graph run 5 times.

```
siyamthanda@ThatsMine:/mnt/c/Users/bheng/OneDrive/Desktop/csc2001f/csc2001f/Assignment2$ make run
python3 src/GenericsKBAVLTest.py
Enter the file name of where all entries are stored: GenericsKB.txt
Enter the file name for all queries: GenericsKB-queries.txt
How many times would you like to run this experiment:100
```

Number of search operations in relation to dataset size
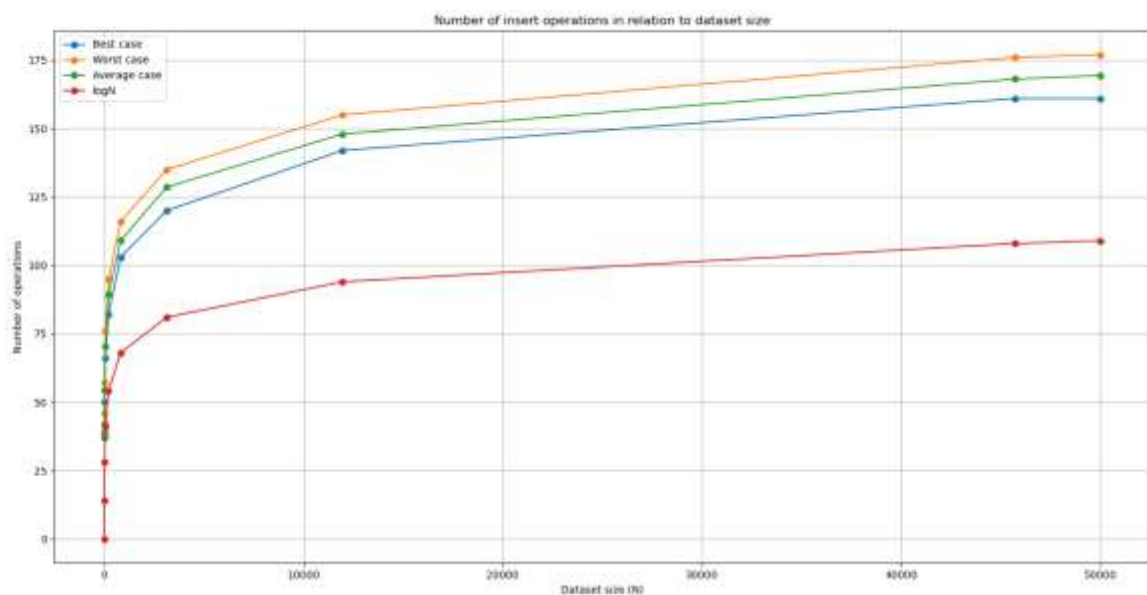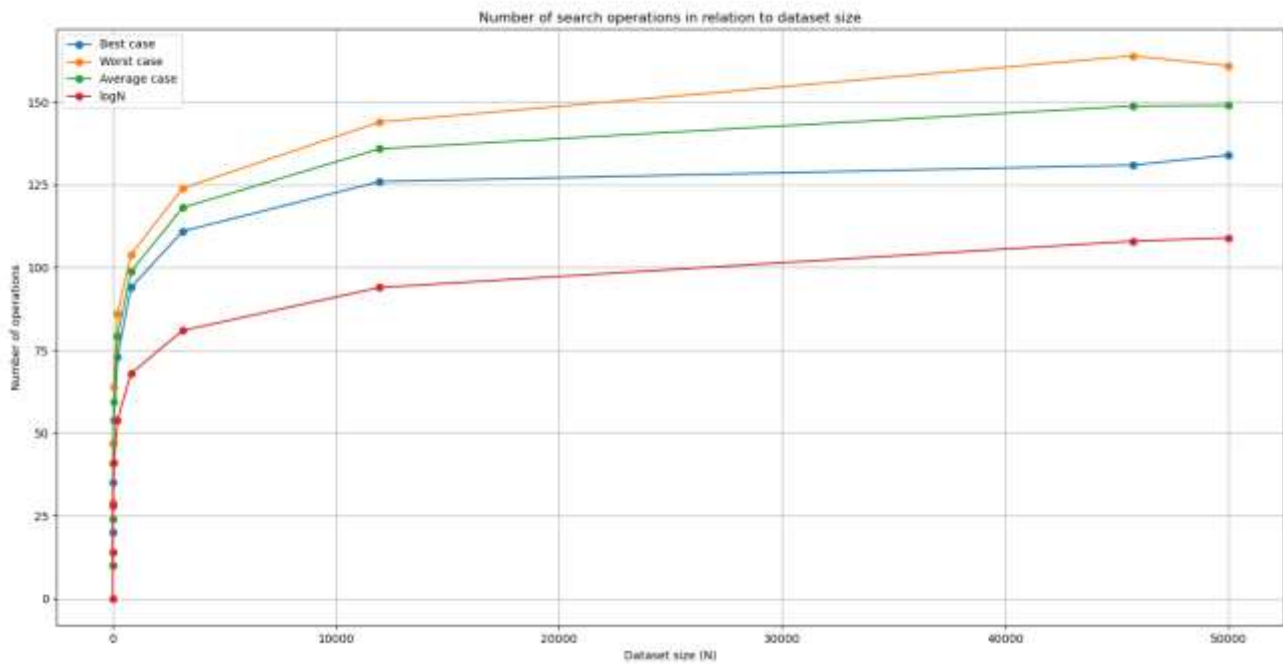
Show above are the graphs for the number of comparisons done for both search and insert for each value of value of n. For graph 1 it is important to note that it is the time taken to insert 10 entries into nodes of size n, not the time taken to populate a tree of size n. For graph 2 it is the time taken to search for 10 queries in a tree of size n. Looking at the general shape of both graphs we can confirm that the shape is indeed logarithmic. Even though our standard log n graph is below the other graphs we know that there exists a value M such that Mg(x) is greater than f(x). In our case our results are all multiples of log(n), therefore we can assert that our big-o for search and insert is O(log n). This makes sense as the constant rebalancing of an AVL tree means ensures that the maximum number if comparisons on the way to the last node logn.

From Results.txt we notice that all of our values are different but the still fall under our upper bound of $\log_2(n) \times 10$ even for the worst case. This tells us that the time complexity for insertions and search is indeed O(log n) as our graphs are multiples of log n. Attached is the results for all the values of n for all 100 runs. There were too many values to put into a table.

Running the program:

Compile the program using **make**. Test part 1 by doing **make test.** In order to run the main program with the graphs type **make run**. The default file for entries is **GenericsKB.txt.** The default file for queries is **GenericsKB-queries.txt**. However you can run the program using any files that are formatted in the same manner. **GenericsKB-additional.txt** is used by the program to generate 10 random entries that are inserted into the entries tree, it should not be deleted. This program requires the downloading of matplotlib therefore doing a **pip install matplotlib** will ensure the graphs are all constructed properly.

Creativity

- Complete automation of the testing process using GenericsKBAVLTest.py.
- This script reads in you GenericsKB and creates multiple samples of different sizes n that are logarithmically spaced apart.
- It will also create a random Entries10 and Querie10 file that is kept standard for each run of the GenericsKbAVLApp.
- It will then run GenericsKbAVLApp using subprocess.run which allows you to input terminal commands using python.
- These files are then passed as args into the main file
- Also automates the process of doing multiple runs and storing an average of all runs for more accurate results
- Incorporated the use of Matplotlib so that graphs are created automatically based on the number of runs accuracy will increase.
- **make test** automatically tests if all the functions of the knowledge base work as intended.
- Results for all runs are stored in Results.txt
- Simple user interface so you can use any files you want for entries and queries as entries have at least 50 000 lines.

Git log

0: commit ce1312452b44c70ba01a4d82f0ae6f631c9cd1e9

1: Author: Siyamthanda Phakathi <phksiy006@myuct.ac.za>

2: Date: Fri Mar 28 16:34:14 2025 +0000

3:

4: Included and refined the ability to compute multiple runs to get more accurate results. Allowed use to dictate the number of runs the want to do

5:

6: commit 1eab87f1e2a3c84f316b96946de96fc0410f26b6

7: Author: Siyamthanda Phakathi <phksiy006@myuct.ac.za>

8: Date: Thu Mar 27 12:17:07 2025 +0000

9:

...

14: Date: Wed Mar 26 03:38:51 2025 +0000

15:

16: Added the ability to generate random sample files of different sizes. Used matplot lib to automate the graph drawing. Used python script to automate the testing process

17:

18: commit ec965af9a94d1ed399f93a293dadacd515335b08

19: Author: Siyamthanda Phakathi <phksiy006@myuct.ac.za>

20: Date: Fri Mar 21 13:50:00 2025 +0000

21:

22: Created my knowledge base for the

23: avl tree. Also implemented the operations counter and the search all function for searching up all queries